GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Scalable Databases With Elasticsearch: An Overview

Anila Ghazanfar

MatrNr: 11351586

Supervisor: Aasish Kumar Sharma

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2025

Abstract

The exponential growth of data from IoT, social media, and real-time applications has driven the need for scalable database solutions capable of handling large-scale workloads efficiently. Traditional databases struggle with these demands, leading to the adoption of distributed and scalable architectures. This report provides an overview of scalable databases, their types, and the motivation behind their adoption. It focuses on Elasticsearch, a distributed search and analytics engine, analyzing its architecture and scalability features that enable efficient data processing.

To evaluate Elasticsearch's scalability, an experimental study was conducted by deploying a cluster and performing load testing. Results demonstrate that as the number of concurrent threads increased from 50 to 200, throughput improved from 1.43×10^{-6} req/sec to 5.74×10^{-6} req/sec, while average response times decreased from 6.54 ms to 5.42 ms. The findings confirm that Elasticsearch effectively scales with workload increases, maintaining low-latency responses and stable performance. These results highlight Elasticsearch's suitability for high-performance computing (HPC) workflows, making it a robust solution for large-scale data analytics and real-time search applications.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 10% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☑ Further, namely: - summarizing research papers

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

1 Introduction

As data volumes continue to grow exponentially, the need for scalable database solutions has become crucial to maintaining performance, reliability, and efficiency in modern applications. Recent estimates indicate that approximately 402.74 million terabytes of data are generated daily [Dur25] and within the past two years, 90% of the world's data has been created. With the rise of platforms like IoT, social media and e-commerce, we are generating data at an unprecedented rate. This growth in data demands system that can scale dynamically and is the major motivator for adopting scalable databases.

1.1 Scalable Databases

Scalability is a fundamental characteristic of modern databases, allowing them to handle growing workloads without compromising performance. It is the ability to grown or shrink resources dynamically as needed. This ability to scale on demand allows businesses to adapt to fluctuating workloads without worrying about downtime or degradation in performance. There are two primary approaches to scalability: **vertical scaling**, which involves upgrading the existing hardware, and **horizontal scaling**, which distributes the load across multiple machines or nodes. The exponential growth of data, driven by big data analytics, the Internet of Things (IoT), artificial intelligence (AI), and real-time applications, has exposed the limitations of traditional databases. Conventional relational databases often struggle to keep pace with increasing data volumes, leading to performance bottlenecks, high operational costs, and reduced efficiency in large-scale environments. As a result, scalable databases have emerged as a critical solution for ensuring seamless data processing, storage, and retrieval.

1.2 Elasticsearch as a Scalable Database

Among various scalable database solutions, **Elasticsearch** has gained significance as a powerful **distributed search and analytics engine**. It is built on **Apache Lucene** and is designed for handling structured and unstructured data efficiently. It plays a crucial role in modern data infrastructures by enabling fast, full-text search, real-time analytics, and distributed data storage. Unlike traditional databases, Elasticsearch excels in scalability due to its **distributed architecture, high-speed indexing, and flexible schema design**. These attributes make it particularly suitable for handling large-scale workloads, such as log analytics, e-commerce search, security monitoring, and High-Performance Computing (HPC) environments.

1.3 Contributions

The contributions of this report are as follows:

- **Exploring Scalable Databases:** An in-depth exploration of scalable databases, their types, key features, and the underlying motivations for their adoption in modern data-driven systems.
- **Analysis of Elasticsearch Architecture:** A comprehensive examination of Elasticsearch's architecture, highlighting its core components and key concepts that enable its scalability and performance.

- **Review of Relevant Literature:** A short review of relevant publications that demonstrate how Elasticsearch is effectively utilized within High Performance Computing (HPC) workflows and large-scale data analysis tasks.
- **Experimental Load Testing:** Conducting experimental load testing of an Elasticsearch cluster to evaluate its performance and scalability under different workloads, providing empirical insights into its capabilities.

The report is organized as follows: Section 2 introduces scalable databases and its types and motivation for adopting scalable databases. Section 3 explores fundamental concepts of Elasticsearch architecture and finally, Section 4 summarizes two publications that have utilized Elasticsearch in HPC workflows demonstrating practical applications, showcasing how Elasticsearch enhances data retrieval, performance monitoring, and analysis in complex computational workflows. Section 5 demonstrate how load testing of a 3 node elastic search cluster is done and finally, results are summarized and discussed in Section 6.

2 Scalable Databases

The traditional databases often struggle to maintain performance under increasing workloads as the volume of data continues to grow exponentially. Scalable databases are designed to handle an increasing amount of data or workload, ensuring efficient data storage, retrieval, and processing without compromising performance, reliability or functionality. These databases achieve scalability through **vertical scaling** (scaling up) by upgrading hardware or **horizontal scaling** (scaling out) by distributing data across multiple servers or nodes. Scalable databases are essential in modern applications such as big data analytics, high-performance computing (HPC), and cloud-based solutions.

2.1 Types of Scalable Databases

This section explores various types of scalable databases.

2.1.1 NewSQL and Distributed SQL Databases

NewSQL databases combine the ACID compliance of SQL with NoSQL scalability, while Distributed SQL databases distribute data across multiple nodes while maintaining SQL capabilities.

- **CockroachDB:** Fault-tolerant, highly available SQL database.
- **VoltDB:** High-speed, in-memory relational database for real-time analytics.
- **TiDB:** MySQL-compatible, auto-scaling distributed SQL database.

2.1.2 NoSQL Databases

Designed for flexible schemas, high availability, and horizontal scaling.

- **Document Stores:** JSON-like document storage (e.g., MongoDB).

- **Column-Family Stores:** Optimized for big data (e.g., Apache Cassandra).
- **Key-Value Stores:** Fast data retrieval (e.g., Redis).

2.1.3 Specialized Databases

- **Time-Series Databases:** Optimized for time-stamped data (e.g., InfluxDB, TimescaleDB).
- **In-Memory Databases:** Store data in RAM for ultra-fast access (e.g., Redis, Memcached).
- **Cloud Databases:** Scalable cloud-native solutions (e.g., Amazon RDS, Google Cloud Spanner).

2.1.4 Elasticsearch as a Scalable Database

Elasticsearch, primarily a search engine, also functions as a distributed database for structured and unstructured data, offering high-speed indexing and real-time analytics.

2.2 Key Features of Scalable Databases

Scalable databases incorporate several key features to manage large workloads with high availability and performance:

1. **Replication:** Data is replicated across nodes for fault tolerance, high availability, and improved read performance.
2. **Partitioning/Sharding:** Large datasets are split into smaller "shards" distributed across nodes, improving performance and enabling horizontal scalability.
3. **Distributed Architecture:** Data and workload are spread across multiple servers to prevent bottlenecks and allow seamless scalability.
4. **High-Performance Storage:** Columnar and in-memory storage (e.g., Apache Cassandra, Redis) enable faster data retrieval and optimized queries.
5. **Interconnect and Load Balancing:** Efficient interconnects and load balancing distribute queries evenly, preventing node overload and optimizing performance.

These features ensure scalability, reliability, and high performance for modern applications.

2.3 Motivation for Adopting Scalable Databases

Key motivations for adopting scalable databases include:

- **Growing Data Demands:** The rise of IoT, social media, and AI-driven applications generates vast amounts of data, requiring databases capable of handling increasing volumes without performance loss.
- **Performance and User Experience:** Modern applications demand low-latency responses and global availability to provide seamless user experiences, as seen in platforms like Netflix and YouTube.

- **Business Continuity and Availability:** Scalable databases ensure high availability and minimize downtime, critical for preventing financial and reputational damage.
- **Challenges of Traditional Systems:** Legacy databases face scalability limits due to vertical scaling, while modern databases use horizontal scaling to distribute workloads and reduce complexity and costs.

Scalable databases enable organizations to future-proof their infrastructure, optimize performance, and meet the growing demands of data-driven applications. This report will explore how Elasticsearch fits into this landscape as a distributed search and analytics engine.

3 Elasticsearch

Elasticsearch is a **distributed, open-source search and analytics engine** built on Apache Lucene, designed to process vast amounts of data and deliver **near real-time full-text search capabilities**. The data is stored as **JSON** documents. Common use cases of Elasticsearch include: full-text search, log analytics, application monitoring, and business intelligence.

3.1 Elasticsearch Architecture

Elasticsearch follows a distributed architecture that allows for scalability, fault tolerance, and high availability. Key components of the architecture as shown in Figure 2 are discussed. The key concepts as in [KRP22] are summarized in the subsequent sub-sections.

3.1.1 Key Components

3.1.1.1 Document: Data in Elasticsearch is stored in the form of JSON documents which is the basic unit of data storage. Documents are equivalent to rows in a relational database table as in figure 1 the employee table rows correspond to JSON documents in the employee index. Each document consists of fields that correspond to columns in a relational database table 1. An index contains one or more documents, and each document, in turn, includes one or more fields. Once a document is added to an Elasticsearch index, an inverted index (3.2.1) is created, making it instantly searchable.

3.1.1.2 Cluster A cluster is a collection of nodes that collectively store all the data and offer federated indexing and search functions. Each cluster is identified by a unique name and contains a designated "master" node that handles administrative tasks.

3.1.1.3 Node A node is a single Elasticsearch instance that stores data and processes queries. It can play one or more roles for workload isolation and scaling:

- **Master Node:** Controls and Manages cluster-wide operations such as creating/deleting indexes and tracking cluster health.
- **Data Node:** Stores indexed data and perform data related operations.

RDBMS	Elasticsearch
Table	Index
Row	JSON Document
Column	attributes of JSON document

Employee Table			
Name	Location	Age	Email
John	Berlin	24	abc@gmail.com
Max	Göttingen	36	def@gwdg.de
Alice	Frankfurt	28	ghi@gmail.com

Employee Index

```
{
"name": "John",
"location": "Berlin",
"Age": "24",
"Email": abc@gmail.com
}

{
"name": "Max",
"location": "Göttingen",
"Age": "36",
"Email": def@gwdg.de
}

{
"name": "Alice",
"location": "Frankfurt",
"Age": "28",
"Email": ghi@gmail.com
}
```

Figure 1: Analogy to RDBMS table

- **Ingest Node:** Preprocesses documents before indexing, including transformation and enrichment.
- **Coordinating Nodes:** Routes request, handles search reduce phase and distribues bulk indexing. All nodes function as coordinating nodes.
- **Alerting Node:** Runs alerting jobs.
- **Machine Learning Nodes:** Runs machine learning jobs.

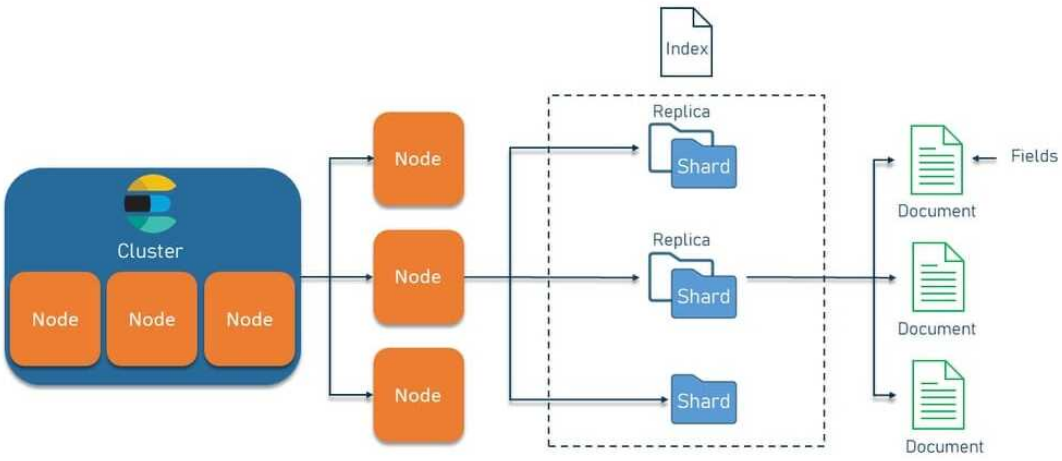

Figure 2: Elasticsearch Architecture

Source: `https://www.altexsoft.com/blog/elasticsearch-pros-cons/`

3.1.1.4 Index An index is a logical grouping of one or more physical shards, with each shard being a Lucene index, which is a self-contained index. It is a collections of documents that share the same structure, similar to tables in relational databases 1, but optimized for search and analytics.

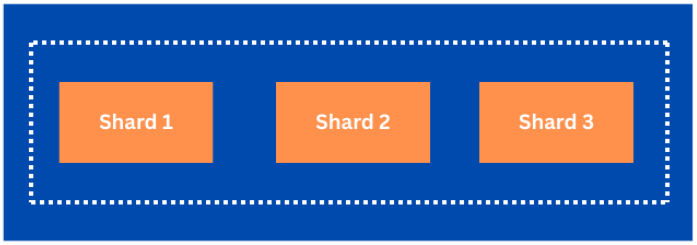

Figure 3: Elasticsearch Index

3.1.1.5 Shard Elasticsearch automatically splits an index into smaller partitions called shards which are then distributed across anumber of nodes. Distributing data across multiple nodes improves performance and scalability. As cluster load increases, Elasticsearch redistributes shards to other clusters, ensuring that data load is balanced. Types of shards in Elasticsearch are:

1. **Primary:** Primary shards store the original copy of the data. The index data is distributed across multiple primary shards. Users can define the number of primary shards and their corresponding replica shards when creating an index.

2. **Replica:** Replicas are copies of primary shards for redundancy and serving data queries that provide high availability and fault tolerance. It allows load balancing of search queries by distributing queries across replica shards. When a node is added or removed, the shards and data are automatically re-balanced. If a node fails, Elasticsearch automatically reassigns shards to other nodes.

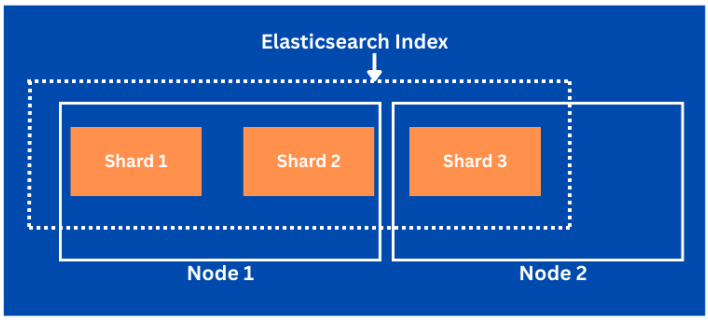

Figure 4: Elasticsearch Shards

3.2 Key Concepts

3.2.1 Inverted index

Elasticsearch uses a data structure called an **inverted index** to store data that maps terms to document locations. It allows for fast full-text search by maintaining term-to-

document mappings, that is, quickly locate documents that include the searched terms. For example, in figure 5, consider a dataset with eight documents, each containing an attribute geo-scope Id. If we were using a forward index, searching for "Europe" would involve scanning the dataset to determine that Europe appears in documents 1, 2, and 7. However, Elasticsearch stores this information as an inverted index, where the term Europe is directly linked to documents 1, 2, and 7. This structure allows Elasticsearch to instantly respond to queries like "Where is Europe?" by immediately returning the relevant document IDs, as the data is inherently organized for fast lookup.

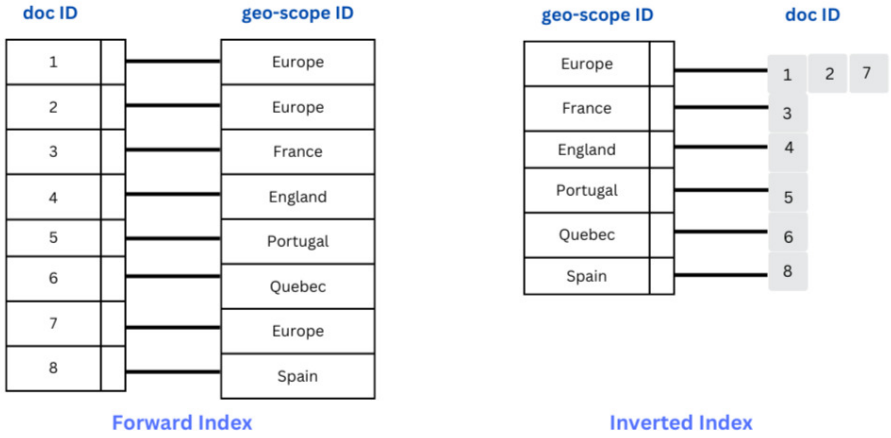

Figure 5: Inverted Index example

3.2.2 Index Template

In Elasticsearch, an index template serves as a blueprint for creating an index. It outlines the structure of documents, specifying the fields, their data types, and any metadata associated with the document type. These configurations are applied during the index creation process and are comparable to a table schema in a relational database. The template includes configurations such as: number of shards and replicas, mapping, priority, etc.

3.2.2.1 Data Mapping It specifies the schema for the documents stored in the index. It can be configured to:

1. **Strict:** With strict mapping, users can define the fields and their data types manually.
2. **Dynamic or Schema on Write:** For ease, dynamic mapping was introduced in Elasticsearch which automatically creates a field mapping when a new field is encountered that hasn't been explicitly defined by the user.
3. **Combined:** Strict and dynamic mapping can be combined, providing flexibility in Elasticsearch's mapping configuration.

3.2.3 Index alias

An index alias is a collection of indices. Documents can be added to an index group through the alias, but only the index designated as the write index can accept document insertions.

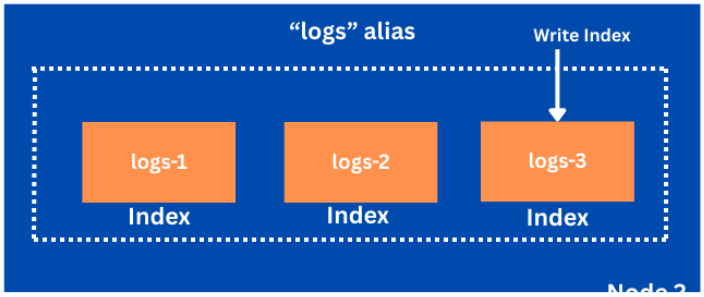

Figure 6: Index Alias

An alias can be defined to encompass all indices that match a specific index pattern (e.g., mylogs-*). Figure 7 illustrates the command that creates an alias named "logs," which groups all indices beginning with "logs-".

```
POST _aliases
{
    "actions": [
        {
          "add": {
                  "index": "logs-*",
                   "alias": "logs"
          }
        }
     ]
}
```

Figure 7: Indices starting with "logs-" are grouped under alias named "logs"

3.2.4 Data stream

A data stream enables the storage of append-only time series data across multiple indices while providing a single named resource for handling requests. Data streams are ideal for logs, events, metrics, and other continuously generated data. Indexing and search requests are submitted directly to a data stream, which automatically routes the request to the **backing indices or hidden indices** that store the data.

Index lifecycle management (ILM) can be used to automate the handling of these backing indices. For example, ILM can automatically move older backing indices to more cost-effective storage and remove unnecessary indices. ILM helps reduce costs and management overhead as data grows. While data can be queried from all indices, it can only be written to the most recent index, as indicated in blue in Figure 8.

3.3 Data Flow in Elasticsearch

Elasticsearch data flows through the following steps as discussed in [Sto25]:

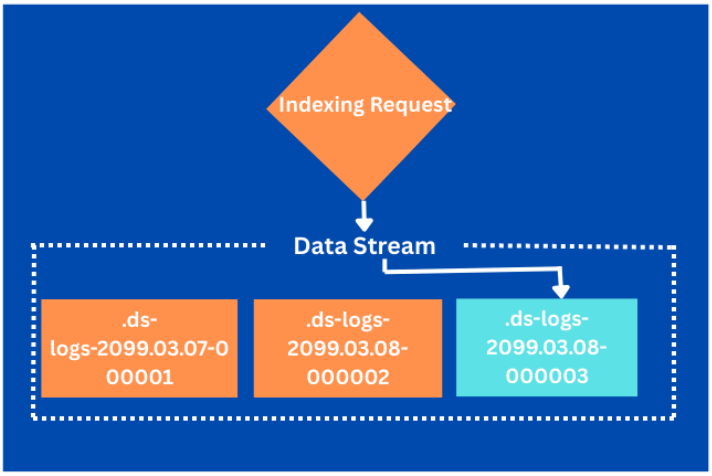

Figure 8: Data Stream

- **Query Submission:** A query is sent via the Elasticsearch API.
- **Coordinator Node:** The API directs the query to a coordinator node, which determines which node or nodes should handle the query.
- **Routing to Shards:** The coordinator identifies the appropriate shards, either on one node or across multiple nodes, where the relevant data is stored.
- **Shard Processing:** The relevant shards retrieve the documents matching the query.
- **Aggregation of Results:** The shards send the results (indices) back to the coordinator.
- **Sorting and Organizing:** The coordinator organizes and sorts the indices received from different shards.
- **Document Retrieval:** Once sorted, the coordinator fetches the actual documents from the relevant shards.
- **Return Data:** Finally, the documents are sent back to the application via the API.

This process allows Elasticsearch to efficiently manage and retrieve large datasets by distributing the load across nodes and shards. The following section will focus on its application in High-Performance Computing (HPC) workflows.

4 HPC workflow leveraging Elasticsearch

Elasticsearch's scalability and efficiency offer significant advantages for handling large datasets in HPC environments, facilitating faster data retrieval, analysis, and overall system performance. This section will examine how Elasticsearch can be effectively integrated into HPC workflows to optimize data processing tasks.

HPC workloads generate massive amounts of data from simulations, logs, and metrics. Elasticsearch provides a scalable, distributed platform for managing and analyzing this data in near real-time. Key use cases of Elasticsearch in HPC workflows include:

- **Log Aggregation:** Elasticsearch allows for centralized collection and searching of logs from multiple HPC nodes, thereby helping in efficient debugging and monitoring of system activities.
- **Performance Monitoring:** Elasticsearch can visualize system metrics such as CPU, memory, and network usage, which are useful to identify and resolve performance bottlenecks in HPC systems.
- **Data Search and Retrieval:** By leveraging Elasticsearch, HPC workflows can perform fast queries over large datasets, such as simulation outputs or experimental results, improving data accessibility.
- **Anomaly Detection and Predictive Analytics:** Elasticsearch enables the integration of machine learning models with indexed data, facilitating real-time anomaly detection and predictive analysis.

The next sub-sections will focus on two relevant publications that explore the integration of Elasticsearch within HPC environments. These studies demonstrate practical applications, showcasing how Elasticsearch enhances data retrieval, performance monitoring, and analysis in complex computational workflows.

4.1 Real-Time I/O-Monitoring of HPC Applications with SIOX, Elasticsearch, Grafana and FUSE (2017)

The framework [BK17] for real-time I/O monitoring of HPC applications is designed to provide insights into the I/O behavior of applications running on high-performance computing systems. It leverages several open-source technologies, including SIOX, Elasticsearch, Grafana, and FUSE, to create a comprehensive monitoring solution. Here's a detailed explanation of each component and how they work together:

- **SIOX (System I/O eXplorer):**
  - Purpose: SIOX acts as a wrapper around applications to instrument their I/O calls. It captures I/O activities generated by applications and creates an activity stream.
  - Functionality: When an application is started with the SIOX wrapper (e.g., SIOX(<exec>)), it intercepts I/O operations from various interfaces such as POSIX, MPI, HDF5, and NetCDF. This allows for detailed tracking of how applications interact with the file system.
- **Elasticsearch:**
  - Role: Elasticsearch serves as the backend data store for the I/O metrics collected by SIOX.
  - Features: It is a distributed, scalable search and analytics engine that allows for real-time indexing and querying of data. This enables quick access to I/O statistics, facilitating immediate analysis and monitoring.

  - Data Structure: Metrics are sent to Elasticsearch in JSON format, allowing for structured storage and efficient retrieval.

- **Grafana:**
  - Visualization Tool: Grafana is used to create interactive dashboards that visualize the data stored in Elasticsearch.
  - User Interface: It provides various widgets (e.g., graphs, tables) to display metrics over time, making it easy for users to monitor application performance and identify trends.
  - Dynamic Features: Grafana supports templating and auto-refreshing, allowing users to customize their dashboards and view real-time data without manual updates.

- **FUSE (Filesystem in Userspace):**
  - Non-Intrusive Monitoring: FUSE is utilized to monitor I/O operations related to virtual memory. It allows the framework to intercept I/O requests made by applications that use the mmap() function without requiring changes to the application code.
  - Mount Points: By mounting directories containing relevant files, FUSE forwards I/O requests from virtual memory to the monitoring framework, ensuring comprehensive data collection.

4.1.1 Data Flow and Interaction

- Application Execution: When an HPC application is executed with the SIOX wrapper, it generates I/O activities that are captured in real-time.
- Data Collection: These activities are aggregated and sent to Elasticsearch, where they are indexed for fast retrieval.
- Visualization: Grafana queries Elasticsearch to visualize the collected metrics, providing users with insights into the I/O behavior of their applications.

4.1.2 Benefits

- **Real-Time Monitoring:** Continuously tracks I/O performance for quick issue detection.
- **Optimized Resource Use:** Identifies inefficiencies to enhance workflow efficiency.
- **User-Friendly Interface:** Grafana enables intuitive I/O metric analysis.

This framework enhances I/O monitoring in HPC by integrating SIOX, Elasticsearch, Grafana, and FUSE, enabling better performance insights and resource optimization.

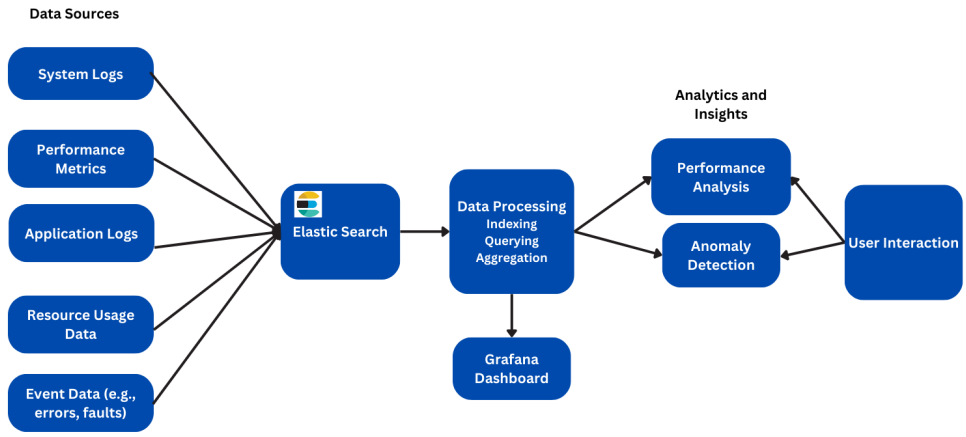

Figure 9: Data Flow in HPC Workflows Utilizing Elasticsearch and Grafana for Performance Monitoring and Analysis

4.2 End-to-end online performance data capture and analysis for scientific workflows (2021)

[Pap+21] present a framework for online performance data capture of scientific workflows. The framework integrates various well-established and newly developed tools to collect, process, and analyze performance data from scientific workflows. It is built around the Pegasus Workflow Management System (WMS) and leverages the ELK stack (Elasticsearch, Logstash, and Kibana) for data management and visualization. The key components are summarized as follows:.

- **Pegasus WMS:** This is the core system that allows users to design and execute scientific workflows. It transforms high-level abstract workflows into executable workflows that can run on distributed computing resources.
- **Data Collection Tools:** The framework extends existing components of Pegasus, such as:
  - pegasus-monitord: Monitors workflow execution and collects performance metrics.
  - pegasus-transfer: Manages data transfers and logs transfer statistics.
- **Data Ingestion:**
  - Logstash: This tool is used to ingest data from various sources, including RabbitMQ, where monitoring data is published. Logstash processes this data and sends it to Elasticsearch for storage.
  - Elasticsearch: Acts as the central repository for all performance data. It allows for efficient storage, indexing, and querying of large volumes of data, enabling users to retrieve specific metrics and logs easily.
  - Kibana: A visualization tool that provides a user-friendly interface for exploring and analyzing the data stored in Elasticsearch. The authors developed a custom Kibana plugin tailored to the needs of monitoring workflow executions, allowing users to create interactive dashboards.

4.2.1 Data Flow

- Monitoring and Data Capture: As workflows execute, pegasus-monitord collects performance metrics (e.g., CPU usage, I/O operations) and publishes this data to RabbitMQ.
- Data Ingestion: Logstash retrieves the data from RabbitMQ and ingests it into Elasticsearch, where it is indexed for efficient querying.
- Real-Time Analysis: Users can query Elasticsearch to retrieve performance metrics in near real-time, allowing for immediate insights into workflow execution.
- Visualization: The custom Kibana plugin enables users to visualize the data through interactive dashboards, making it easier to identify performance issues and trends.

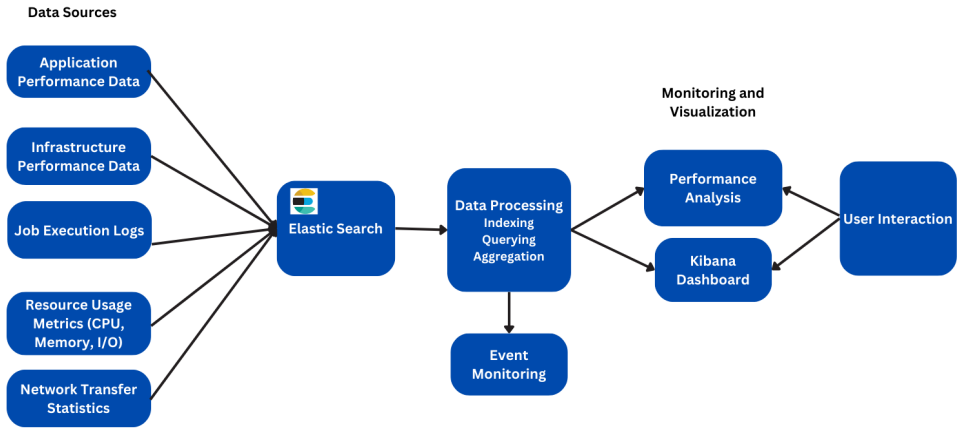

Figure 10: Diagram illustrating the end-to-end framework for online performance data capture and analysis in scientific workflows

4.2.2 Benefits

- **Comprehensive Monitoring:** Captures data from multiple sources (network, filesystem, compute) for a holistic workflow view.
- **Timely Insights:** Near real-time monitoring helps detect and resolve issues instantly.
- **Advanced Analysis:** Elasticsearch and Kibana enable powerful querying and visualization for deeper insights.
- **Machine Learning Integration:** Collected data aids in training ML models for workflow optimization.

This framework enhances scientific workflow monitoring by integrating robust data collection, storage, and visualization for improved HPC performance.

5 Experiment

To evaluate the scalability of an Elasticsearch cluster, a load-testing experiment was conducted using Apache JMeter. The cluster consists of three nodes: one master node and two data nodes with roles as data and ingest. Docker Compose was used to set up the entire environment, including Elasticsearch, Kibana, and Apache JMeter. The configuration file (docker-compose.yml) is provided in the appendix section A.

5.1 Objective

The goal of this experiment is to analyze the scalability and efficiency of a multi-node Elasticsearch cluster under increasing query loads. The results will help determine how well the cluster handles concurrent search requests and whether any performance bottlenecks arise as the load increases.

5.2 Experimental Setup

- **Cluster Configuration:** 3-node Elasticsearch cluster (1 master node, 2 data nodes with roles: data, ingest)
- **Testing Tool:** Apache JMeter (section)
- **Deployment:** Docker Compose
- **Request Type:**HTTP POST requests
- **Elasticsearch API Used:** _search
  - Query Used:

```
{
    "query": {
                "match_all": {}
            }
}
```

- **Index Used:** kibana_sample_data_ecommerce
- **Thread Groups:** 50, 100, 150, 200 concurrent users
- **Metrics Collected:** Throughput, average response time, 90th percentile response time, and error rate

5.3 Performance Metrics Measured

The performance metrics measured are as follows:

- **Throughput (Requests per Second):** Measures how many queries the cluster can handle concurrently.
- **Average Response Time (ms):** The average time taken to return search results.
- **90th Percentile Response Time (ms):** The response time below which 90% of requests are completed.
- **Error Rate (%):** Percentage of failed requests.

The load tests were designed to analyze the cluster's response under increasing workloads by measuring how throughput and response times scale as the number of concurrent users increases. The obtained results provide insights into the cluster's scalability and performance characteristics.

6 Results

JMeter scalability test results in Table 1 show how the Elasticsearch cluster performed under increasing loads. These results are shown graphically in Figure 11. Let's analyze the key metrics:

- **Throughput (Requests per Second)** increases with the number of threads. This suggests that as more concurrent users (threads) send requests, the system is able to process them at a higher rate. However, the increase is not strictly linear, indicating some performance limitations.
- The **average response time** decreases as load increases. Typically, response time increases with load, but here, the variation might suggest effective load balancing or caching effects in Elasticsearch.
- **90th Percentile Response Time (ms)** measures how fast 90% of requests are served. A slight decrease at 200 threads suggests the cluster might be handling the load efficiently, possibly due to caching mechanisms.
- **Error Rate (%)** is 0% for all test cases → Excellent stability; no failed requests.

In conclusion, Elasticsearch is scaling well with increasing threads. No errors show that the cluster is stable. Throughput scales but not linearly, indicating potential bottlenecks or internal optimizations at play. Lower response times at higher loads could be due to caching, efficient query execution, or other optimizations.

7 Conclusion

This report explored scalable databases, their key features, and their significance in handling modern data-intensive applications. It examined Elasticsearch's architecture, its

Threads	Throughput (req/sec)	Avg Response (ms)	90th%ile (ms)	Error Rate (%)
50	1.43E-06	5.2216	7	0
100	2.87E-06	6.5364	8	0
150	4.30E-06	5.7492	7	0
200	5.74E-06	5.4254	6	0

Table 1: Scalability Results

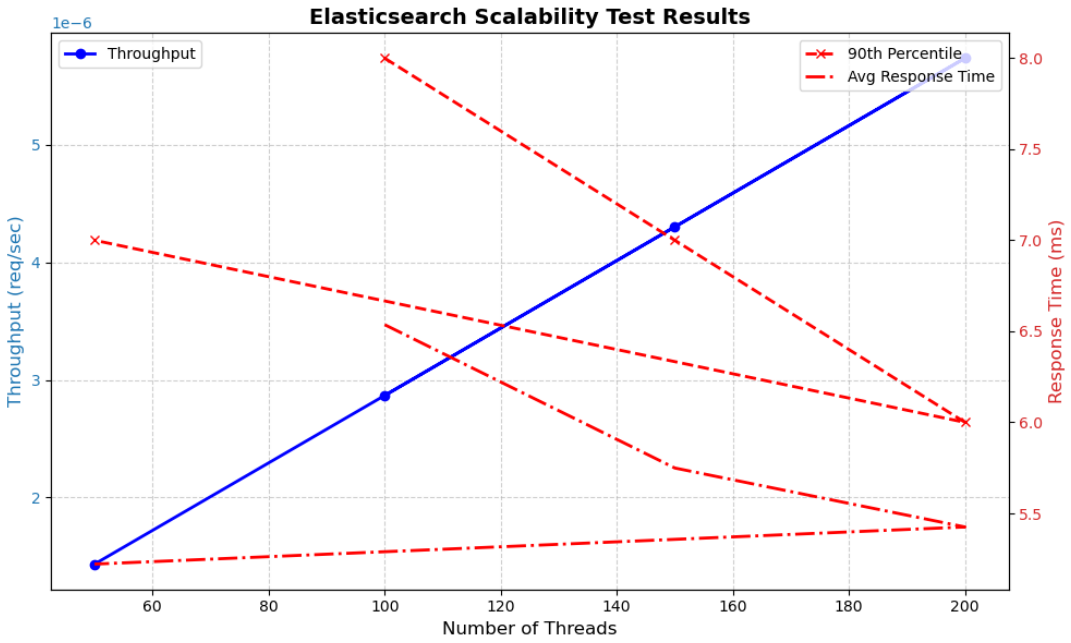

Figure 11: Scalability Results Plot

role in HPC workflows, and conducted experimental scalability testing on an Elasticsearch cluster.

The scalability results demonstrate that as the number of threads increases, throughput improves, and response times show a decreasing trend. Specifically, throughput increased from 1.43×10^{-6} req/sec at 50 threads to 5.74×10^{-6} req/sec at 200 threads. Meanwhile, the average response time improved from 6.54 ms at 100 threads to 5.42 ms at 200 threads, and the 90th percentile latency decreased from 8 ms to 6 ms, indicating better request handling under higher loads.

These findings validate Elasticsearch's capability to scale efficiently while maintaining low response times, making it a viable choice for HPC and data-intensive applications. Future work can further analyze its performance under varying workloads and optimize indexing strategies for enhanced scalability.

References

[BK17] Eugen Betke and Julian Kunkel. "Real-time I/O-monitoring of HPC applications with SIOX, elasticsearch, Grafana and FUSE". In: *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^ 3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers 32*. Springer. 2017, pp. 174–186.

[Dur25] F. Durate. *Amount of Data Created Daily (2024)*. `https://explodingtopics.com/blog/data-generated-per-day`. Accessed: 2025-3-19. 2025.

[KRP22] N. Kathare, O. V. Reddy, and V. Prabhu. "A Comprehensive Study of Elastic Search". In: *Journal of Research in Science and Engineering (JRSE)* 4 (Nov. 2022). URL: `https://www.bryanhousepub.org/src/static/pdf/JRSE-2022-4-11_7.pdf`.

[Pap+21] George Papadimitriou et al. "End-to-end online performance data capture and analysis for scientific workflows". In: *Future Generation Computer Systems* 117 (2021), pp. 387–400.

[Sto25] Pure Storage. *What is Elasticsearch Architecture?* `https://www.purestorage.com/de/knowledge/elasticsearch-architecture.html`. Accessed: 2025-3-20. 2025.

A Experimental setup using Docker-Compose

The configuration details for setting up Elasticsearch, Kibana, and Apache JMeter are provided in the "docker-compose.yml" file in the long Listing 1. This file should be placed in a designated directory. To start the services, navigate to that directory and execute the following command:

```
sudo docker-compose up -v
```

Listing 1: Start Services

This setup ensures that all components are deployed in a containerized environment, maintaining consistency and ease of replication across different systems.

Use the following command to stop the Elasticsearch cluster, Kibana, and JMeter, removing all associated volumes:

```
sudo docker-compose down -v
```

Listing 2: Stop Services

Listing 1: docker-compose.yml

```
version: "3.7"
services:
  es01:
    image: "docker.elastic.co/elasticsearch/elasticsearch-oss:7.10.2"
    mem_limit: 8g  # Container-wide memory limit
    ports:
      - "9200:9200"
      - "9300:9300"
    environment:
      node.name: es01
      cluster.name: mycluster
      node.roles: master  # Master-only node
      discovery.seed_hosts: es02,es03
      cluster.initial_master_nodes: es01
      network.host: 127.0.0.1
      http.host: 0.0.0.0     # Explicit HTTP binding
      transport.host: 0.0.0.0 # Explicit transport binding
      bootstrap.memory_lock: "true"
      ES_JAVA_OPTS: -Xms4g -Xmx4g  # 4GB fixed heap
    volumes:
      - "es-data-es01:/usr/share/elasticsearch/data"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    healthcheck:
      test: ["CMD-SHELL", "curl -sSf http://localhost:9200/_cluster/health?local=true || exit 1"]
      interval: 10s
      timeout: 30s
      retries: 30
      start_period: 60s
    networks:
      - esnet

```

```
  es02:
    image: "docker.elastic.co/elasticsearch/elasticsearch-oss:7.10.2"
    mem_limit: 8g  # Container-wide memory limit
    ports:
      - "9201:9200"
      - "9301:9300"
    environment:
      node.name: es02
      cluster.name: mycluster
      node.roles: data, ingest  # No master role
      discovery.seed_hosts: es01,es03
      #cluster.initial_master_nodes: es01,es02,es03
      network.host: 127.0.0.1
      http.host: 0.0.0.0     # Explicit HTTP binding
      transport.host: 0.0.0.0 # Explicit transport binding
      bootstrap.memory_lock: "true"
      ES_JAVA_OPTS: -Xms4g -Xmx4g  # 4GB fixed heap
    volumes:
      - "es-data-es02:/usr/share/elasticsearch/data"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    healthcheck:
      test: ["CMD-SHELL", "curl -sSf http://localhost:9200/_cluster/health?local=true || exit 1"]
      interval: 10s
      timeout: 30s
      retries: 30
      start_period: 60s
    networks:
      - esnet

  es03:
    image: "docker.elastic.co/elasticsearch/elasticsearch-oss:7.10.2"
    mem_limit: 8g  # Container-wide memory limit
    ports:
      - "9202:9200"
      - "9302:9300"
    environment:
      node.name: es03
      cluster.name: mycluster
      node.roles: data, ingest  # No master role
      discovery.seed_hosts: es01,es02
      #cluster.initial_master_nodes: es01,es02,es03
      network.host: 127.0.0.1
      http.host: 0.0.0.0     # Explicit HTTP binding
      transport.host: 0.0.0.0 # Explicit transport binding
      bootstrap.memory_lock: "true"
      ES_JAVA_OPTS: -Xms4g -Xmx4g  # 4GB fixed heap
    volumes:
      - "es-data-es03:/usr/share/elasticsearch/data"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    healthcheck:
      test: ["CMD-SHELL", "curl -sSf http://localhost:9200/_cluster/health?local=true || exit 1"]
      interval: 10s
      timeout: 30s
      retries: 30
      start_period: 60s
    networks:
      - esnet

  kibana:
    image: "docker.elastic.co/kibana/kibana-oss:7.10.2"
    depends_on:
      es01:
        condition: service_healthy
```

```
                es02:
                  condition: service_healthy
                es03:
                  condition: service_healthy
            ports:
              - "5601:5601"
            environment:
              - 'ELASTICSEARCH_HOSTS=["http://es01:9200","http
                  ://es02:9200","http://es03:9200"]'
              - logging.verbose=false  # Reduce log verbosity
              - server.defaultRoute=/app/home  # Skip the
                  redirect
            healthcheck:
              test: ["CMD-SHELL", "curl -sSf http://localhost
                  :5601/api/status > /dev/null || exit 1"]
              interval: 30s
              timeout: 10s
              start_period: 60s
            networks:
              - esnet
            mem_limit: 1g  # Add memory limit

          jmeter:
            image: "justb4/jmeter:latest"
            container_name: jmeter
            volumes:
              - ./jmeter-tests:/tests
            networks:
              - esnet
            #network_mode: "bridge"
            environment:
              - JVM_ARGS=-Xms4g -Xmx8g  # 4GB min, 8GB max
              - ES_HOST=es01  # Consistent host reference
            command:
              - -n
              - -t /tests/es_benchmark_new.jmx
              - -Jhostname=es01  # Pass hostname as property
              - -l /tests/results_200.jtl
              - -j /tests/jmeter.log
            healthcheck:
              test: ["CMD-SHELL", "curl -f http://es01:9200/
                  _cluster/health?wait_for_status=yellow || exit
                   1"]
              interval: 30s
              timeout: 60s
              retries: 20
            depends_on:
              es01:
                condition: service_healthy
              es02:
                condition: service_healthy
              es03:
                condition: service_healthy

        volumes:
          es-data-es01:
            driver: local
          es-data-es02:
            driver: local
          es-data-es03:
            driver: local

        networks:
          esnet:
            name: esnet  # Explicitly name the network
            driver: bridge
```

B Adding sample data to Kibana

- Access Kibana at http://localhost:5601

- Navigate to: Home → Try sample data → Sample eCommerce orders → Add data
- Verify the index was created using the following command:

```
sudo curl -X GET "http://localhost:9200/kibana_sample_data_ecommerce/_count?pretty"
```

Listing 3: Verify Index

The output will be like:

```
{
  "count" : 4675,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  }
}
```

Listing 4: Verify Index Output

C JMeter Test Plan

C.1 JMeter GUI

- Open JMeter GUI by running the command in the directory /apache-jmeter-5.6.3/bin.
  The GUI can be downloaded from binaries section in https://jmeter.apache.org/download_jmeter.

```
./jmeter.sh  # Linux/Mac | or jmeter.bat on Windows
```

Listing 5: Start Jmeter GUI

C.2 Create a Test Plan:

- Right-click Test Plan → Add → Thread Group.
- Configure:
  - Number of Threads (Users): 50 (adjust as needed)
  - Ramp-up Period: 25 seconds
  - Loop Count: 100

C.2.1 Add HTTP Request Sampler:

- Right-click Thread Group → Add → Sampler → HTTP Request.
- Configure:
  - Protocol: http
  - Server Name: es01 (Docker service name)
  - Port: 9200
  - Path: /kibana_sample_data_ecommerce/_search (adjust index name)
  - Method: POST
  - Body Data:

```
{
    "query": {
                "match_all": {}
            }
}
```

Listing 6: Post Request: Body Data

C.2.2 Add Listeners (for results):

- Right-click Thread Group → Add → Listener → View Results Tree.
- Add another listener: Summary Report.

C.3 Save the Test Plan:

Save as es_benchmark_new.jmx in ./jmeter-tests/ (mounted volume).

C.4 Running the Test

- Start Services: docker-compose up -d
- JMeter Automatically Executes:

  The test plan (es_benchmark_new.jmx) runs and saves results to ./jmeter-tests/results_50.jtl. results_50.jtl is the name specified in docker-compose service jmeter.
- Analyze Results: Check ./jmeter-tests/results.jtl (CSV) or use JMeter's GUI to load the file.